# Modeling the Risk of Software Faults

Prepared for

NASA Independent Verification and Validation Facility

FAU Technical Report TR-CSE-00-06

Taghi M. Khoshgoftaar*
Edward B. Allen
Florida Atlantic University
Boca Raton, Florida USA

February 2000

*Readers may contact the authors through Taghi M. Khoshgoftaar, Empirical Software Engineering Laboratory, Dept. of Computer Science and Engineering, Florida Atlantic University, Boca Raton, FL 33431 USA. Phone: (561)297-3994, Fax: (561)297-2800, Email: taghi@cse.fau.edu, URL: www.cse.fau.edu/esel/.

**Executive Summary**

Development teams apply various techniques to improve software reliability, such as independent verification and validation (IV&V), reengineering, extra reviews, additional testing, and strategic assignment of personnel. Due to resource and time constraints, one must often target reliability enhancement activities to high-risk modules. *Software quality models* predict which modules should be targeted.

The product of a software quality model is predictions. For example, a model could be designed to do one of the following.

- Predict membership in fault-prone class for each module
- Predict the number of faults expected in each module

With predictions in hand, project managers can prioritize and target software enhancement activities toward those modules that need improvement the most.

Cost-effective software quality models must be developed through a disciplined methodology that balances accuracy with practical data collection. Drawing on experience with data from a variety of software development organizations, our methodology is based on the following principles.

- Measure the past to predict the future.
- Exploit your gold mines.
- Software metrics are candidate predictors.
- Linear models are not enough.
- Empirical validation must be realistic.

This report provides an explanation of each principle, illustrated by case studies by the Empirical Software Engineering Laboratory at Florida Atlantic University.

*Keywords*: software reliability, faults, fault-prone modules, software metrics, classification, regression models, multiple linear regression, curvilinear regression

# 1 Introduction

Development teams apply various techniques to improve software reliability, such as independent verification and validation (IV&V), reengineering, extra reviews, additional testing, and strategic assignment of personnel. Due to resource and time constraints, one must often target reliability enhancement activities to high-risk modules. *Software quality models* predict which modules should be targeted.

The product of a software quality model is predictions. For example, a model could be designed to do one of the following.

- Predict membership in fault-prone class for each module
- Predict the number of faults expected in each module

With predictions in hand, project managers can prioritize and target software enhancement activities toward those modules that need improvement the most.

Cost-effective software quality models must be developed through a disciplined methodology that balances accuracy with practical data collection. Drawing on experience with data from a variety of software development organizations, our methodology is based on the principles presented in the following sections, and illustrated by case studies performed by the Empirical Software Engineering Laboratory at Florida Atlantic University.

# 2 Measure the past to predict the future

Software development is inherently a people-intensive enterprise, and software quality is influenced by many factors that vary tremendously among organizations. To achieve

useful accuracy, software quality models must be calibrated for each specific development environment [29]. This is achieved through the following steps.

1. Analyze historical project where reliability is known.

2. Predict current project where reliability is in future.

3. Target reliability enhancement activities now, e.g., IV&V.

A case study acquires historical data on one or more projects. We construct models that could have been developed during the historical project, and calculates assessments that could have been made. The accuracy of those assessments is then evaluated against actual experience. This gives us confidence in predictions for a current project. The return-on-investment (ROI) of using a model can be calculated based on the expected costs of enhancement activities initiated by using the model, the effectiveness of finding faults, and the expected costs of not discovering faults early [14].

# 3 Exploit your gold mines

Many software development organizations have very large databases for project management, configuration management, and problem reporting which capture data on source code and individual events during development. We have found that these databases do contain indicators of which modules will likely have operational faults [15]. Such data bases are *gold mines* for software quality modeling.

Our approach to software quality modeling is aptly described as *data mining* [15], especially when operational faults are rare. *Data mining* is most appropriate when one

seeks valuable bits of knowledge in large amounts of data collected for some other purpose, and when the amount of data is so large that manual analysis is not possible.

A recent status report [27] on the field of software measurement highlights gaps between current research and practice. For example, practitioners want accurate, timely predictions of which modules have high risk, but researchers have yet to find adequate, widely applicable measures and models. Faults are a result of mistakes or omissions by developers, and relevant human behavior in the workplace is notoriously difficult to measure directly.

We take a more pragmatic approach. We capture relevant variation among modules with practical metrics, even though the underlying human behavior is not well understood. Instead of expensive, specialized data collection, we leverage existing databases collected for other purposes, so that the marginal cost of data collection is modest. Rather than waiting for researchers to formulate a general theory, we achieve useful accuracy by empirically calibrating models to each local development environment.

Fayyad [3] defines *knowledge discovery in databases* as "the nontrivial process of identifying valid, novel, potentially useful, and ultimately understandable patterns in data." Fayyad restricts the term *data mining* to one step in the knowledge-discovery process, namely, extracting patterns or fitting models from data. Others use the term more broadly. "Primary data analysis" in statistics is motivated by a particular set of questions that are formulated before acquiring the data. In contrast, data mining analyzes data that has been collected for some other reason. Hand [7] defines data mining as "the process of secondary analysis of large databases aimed at finding unsuspected relationships which are of interest or value to the database owners."

Given a set of large databases or a data warehouse, Fayyad et al. give a framework

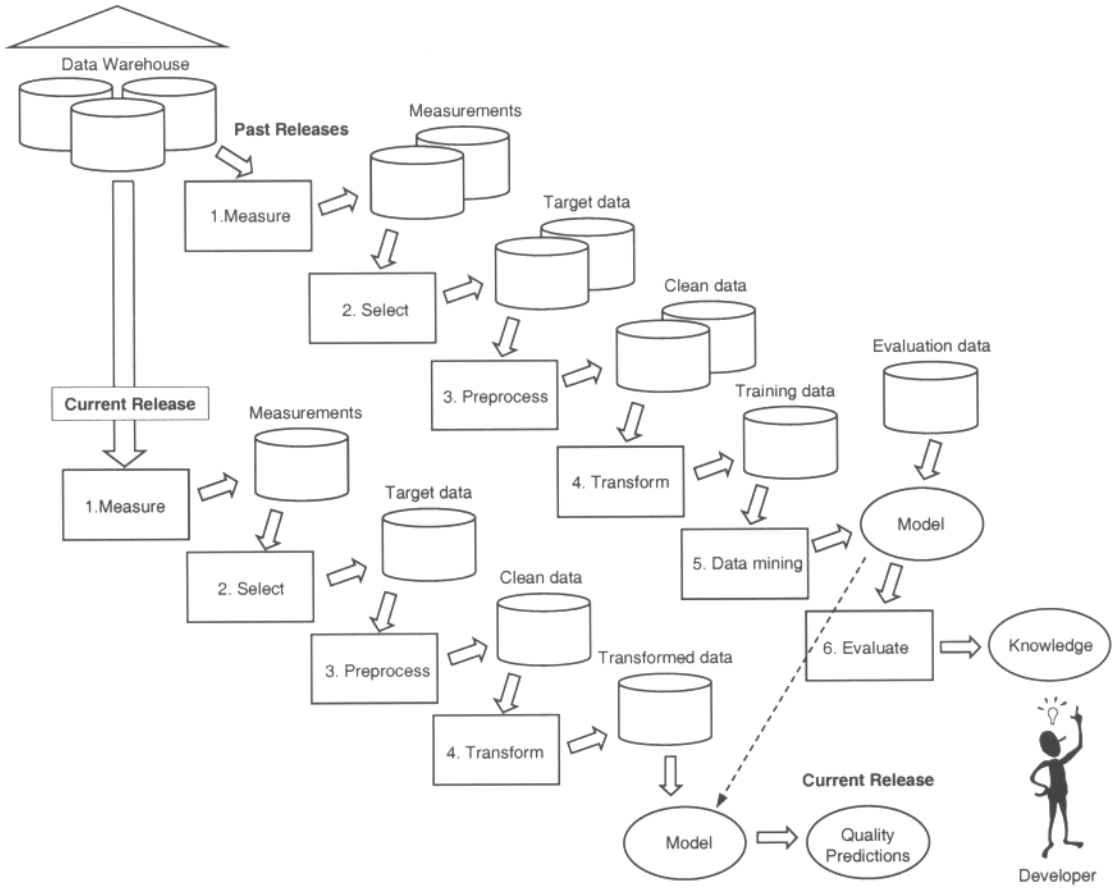


Figure 1: Exploit your gold mines

of major steps in the knowledge-discovery process [4]: (1) selection and sampling of data; (2) preprocessing and cleaning of data; (3) data reduction and transformation; (4) data mining; and (5) evaluation of knowledge. We apply Fayyad's framework to predicting software quality from software development databases. Our framework is shown in Figure 1.

Figure 1 has two similar tracks of processing steps. The upper track processes data on past releases where fault data is known. The results of this track are an empirical model, an assessment of its accuracy, and an interpretation of its structure. The lower track processes data on a current release that is still under development, predicting which modules will be fault-prone through the empirical model. The human figure in the corner represents a developer who will make use of the predictions, the expected accuracy, and the knowledge derived from the model's structure.

In Figure 1, the *Data Warehouse* represents software development databases, such as configuration management systems and problem reporting systems, irrespective of the storage system implementation. A *configuration management system* is an information system for managing multiple versions of artifacts produced by software development processes. For example, most configuration management systems support storage and retrieval of versions of source code. Other features may regulate changes to source code, so that team members do not interfere with each other, and record the history of changes for later review. A *problem reporting system* is an information system for managing software faults from initial discovery through distribution of fixes. In other words, it records events in the debugging process. Most developers of large software products use such systems.

The first step measures available software development databases to derive variables from source code, configuration management transactions, and problem reporting transactions for one or more past releases. Step 2, Select, chooses data for study, resulting in target data. Step 3, Preprocess, accounts for missing data and outliers in the target data, resulting in clean data. Step 4, Transform, may extract features from the clean data, and may transform data for improved modeling. The result is separate transformed

data sets for training and for evaluation. Step 5, Data mining, builds a model based on the training data. Step 6, Evaluate, assesses the model's accuracy using the evaluation data, and analyzes the model's structure.

# 4 Software Metrics

## 4.1 One metric is not enough

Much of the literature on software metrics is aimed to demonstrate the value of individual metrics. However, this does not fulfill our purpose: to build industrial-strength quality models. Our experience with modeling empirical data from industry has indicated that a model with one software metric as the only independent variable does not have useful accuracy and robustness. Lines of code is not enough. McCabe cyclomatic complexity is not enough. The metric that is most highly correlated to faults is not enough.

Our recent case studies have demonstrated that multiple independent variables give more accurate results than models with just one independent variable [10].

The cost of collecting many metrics from source code (or other software product), rather than just a few, is not a practical problem for conventional metrics, because a metric-analyzer software tool is capable of measuring many metrics in one pass. We have found it is more effective to begin with many metrics, and then to apply data mining techniques to choose those with statistically significant empirical relationships to faults.

Pragmatic considerations usually determine the set of available predictors. We do not advocate a particular set of software metrics for software quality models to the exclusion of others recommended in the literature. Because marginal data collection

Table 1: Example code analyzers

| | |
|---|---|
| McCabe tool | McCabe Asso. |
| Logiscope | Verilog S.A. |
| Datrix | Bell Canada |
| Metrics Analyzer for C | FAU |

costs are modest, we prefer to apply data mining to a large well-rounded set of metrics rather than limit the collection of software metrics according to predetermined research questions.

Table 1 lists examples of source code analyzers that can measure many metrics in one pass through the code. The Empirical Software Engineering Laboratory at Florida Atlantic University has Logiscope, Datrix and its own Metrics Analyzer for C, for use in software engineering research.

## 4.2 Code metrics are not enough

Failures are the combined result of the difficulty of the implementation job (product attributes), the process applied to the job, and the usage of the product during operations. These factors are rarely uniform over all modules. Code metrics measure only attributes of the *product*, and thus, in most situations, this one dimension is not enough for accurate robust software quality models.

The development histories of modules often differ radically. For example, modules from early releases have been used or tested more than recently developed modules. A stable module may have been developed by only one person, while other modules may have been modified by many different programmers. Indicators of such variations can

Table 2: Abstractions

| |
|---|
| Product: |
| Statements |
| Call graph |
| Control flow graph |
| O-O graphs |
| Process: |
| Reuse |
| Fault history |
| Staff experience or surrogate |
| Discovery: |
| Operational profile or surrogate |

significantly improve model accuracy and robustness.

For example, our case studies have shown that a simple indicator of reuse from a prior release can be a significant independent variable in both classification and regression models [18].

A case study of the Joint Surveillance Target Attack Radar System, JSTARS [13], showed that the likelihood of discovering additional faults during integration and test in a spiral life cycle can be usefully modeled as a function of the module history prior to integration. In other words, in this case, process-related measures derived from configuration management data and problem reporting data were adequate for software quality modeling, without resorting to software product measurement tools and expertise.

Table 2 lists various abstractions that are the basis for measurement of product, process, and usage metrics. Table 3 summarizes the candidate predictors used in a recent classification case study [16]. This case study illustrates that collecting many predictor metrics is indeed practical.

Table 3: A Classification Case Study

| | Candidate predictors: |
|---|---|
| 24 | Product metrics: |
| | — Call graph metrics |
| | — Control flow graph metrics |
| | — Statement metrics |
| 14 | Process metrics |
| 4 | Execution metrics |

# 5 Linear models are not enough

In many cases, curvilinear models represent fault data better than linear models. For example, Morgan and Knafl [25] modeled operational faults in a set of UNIX utility programs based on various product and process metrics. They found that quadratic terms and cross-product terms were significant. In other words, a multivariate linear model was less appropriate for fault data in this case study than a curvilinear model.

In this section, we summarize two of our case studies which illustrate that linear modeling is often not appropriate for real-world software metrics and software quality data.

## 5.1 Classification-Tree Case Study

**System description.** We conducted a case study of a very large legacy telecommunications system [15]. This embedded computer application included numerous finite state machines. Such systems require very high software reliability. A *module* consisted of a set of related source code files. The software was written in a high level language using the procedural development paradigm, and was maintained by professional programmers

Table 4: Metrics Summary

| | Candidate predictors: |
|---|---|
| | Product metrics: |
| 2 | Call graph metrics |
| 13 | Control flow graph metrics |
| 9 | Statement metrics |
| 15 | Process metrics |
| 1 | Deployment metric |

in a large organization. The entire system had significantly more than ten million lines of code. We studied data on new and updated modules in one release of the software.

This case study focused on faults discovered by customers after release as the software quality factor. A module was considered *fault-prone* if any faults were discovered by customers, and *not fault-prone* otherwise.

Faults discovered in deployed systems are typically extremely expensive, because, in addition to down-time due to failures, visits to customer sites are usually required to repair them. Fault data was collected at the module-level by the problem reporting system.

Table 4 summarizes the metrics used in this case study, and Table 5 through Table 9 give detailed definitions. Some process metrics were available at end of beta testing. Consequently, the predictions could be useful for planning reengineering in next release.

**Classification-tree modeling.** Figure 2 shows the classification tree generated by CART [1] in this case study. See Appendix D for details on CART. A tree represents an algorithm that classifies a module. Beginning at the top, an attribute of the module is

Table 5: Call Graph Metrics

| Symbol | Description |
|---|---|
| *CALUNQ* | Number of distinct procedure calls to others. |
| *CAL2* | Number of second and following calls to others. $CAL2 = CAL - CALUNQ$ where $CAL$ is the total number of calls. |

Table 6: Control Flow Graph Metrics

| Symbol | Description |
|---|---|
| *CNDNOT* | Number of arcs that are not conditional arcs. |
| *IFTH* | Number of non-loop conditional arcs, i.e., if-then constructs. |
| *LOP* | Number of loop constructs. |
| *CNDSPNSM* | Total span of branches of conditional arcs. The unit of measure is arcs. |
| *CNDSPNMX* | Maximum span of branches of conditional arcs. |
| *CTRNSTSM* | Total control structure nesting. |
| *CTRNSTMX* | Maximum control structure nesting. |
| *KNT* | Number of knots. A "knot" in a control flow graph is where arcs cross due to a violation of structured programming principles. |
| *NDSINT* | Number of internal nodes (i.e., not an entry, exit, or pending node). |
| *NDSENT* | Number of entry nodes. |
| *NDSEXT* | Number of exit nodes. |
| *NDSPND* | Number of pending nodes, i.e., dead code segments. |
| *LGPATH* | Base 2 logarithm of the number of independent paths. |

Table 7: Statement Metrics

| Symbol | Description |
|---|---|
| *FILINCUQ* | Number of distinct include files. |
| *LOC* | Number of lines of code. |
| *STMCTL* | Number of control statements. |
| *STMDEC* | Number of declarative statements. |
| *STMEXE* | Number of executable statements. |
| *VARGLBUS* | Number of global variables used. |
| *VARSPNSM* | Total span of variables. |
| *VARSPNMX* | Maximum span of variables. |
| *VARUSDUQ* | Number of distinct variables used. |

Table 8: Software Process Metrics

| Symbol | Description |
|---|---|
| *DES_PR* | Number of problems found by designers |
| *BETA_PR* | Number of problems found during beta testing |
| *TOT_FIX* | Total number of problems fixed |
| *DES_FIX* | Number of problems fixed that were found by designers |
| *BETA_FIX* | Number of problems fixed that were found by beta testing in the prior release. |
| *CUST_FIX* | Number of problems fixed that were found by customers in the prior release. |
| *REQ_UPD* | Number of changes to the code due to new requirements |
| *TOT_UPD* | Total number of changes to the code for any reason. |
| *REQ* | Number of distinct requirements that caused changes to the module |
| *SRC_GRO* | Net increase in lines of code |
| *SRC_MOD* | Net new and changed lines of code |
| *UNQ_DES* | Number of different designers making changes |
| *VLO_UPD* | Number of updates to this module by designers who had 10 or less total updates in entire company career. |
| *LO_UPD* | Number of updates to this module by designers who had between 11 and 20 total updates in entire company career |
| *UPD_CAR* | Number of updates that designers had in their company careers |

Table 9: Deployment Metric

| Symbol | Description |
|---|---|
| *USAGE* | Deployment percentage of the module. |

evaluated at each decision node (diamond). The associated threshold determines which edge is taken to the next step. When the algorithm arrives at a leaf (circle), the module is classified according to the label of the leaf.

Nodes 6 and 7 illustrate the nonlinear character of the data in this case study. Both nodes evaluate the number of distinct file-includes (*FILINCUQ*). If $FILINCUQ \leq 29$, then the module is predicted to be fault-prone; if $29 < FILINCUQ \leq 34$, then the module is predicted to be not fault-prone; and if $34 < FILINCUQ$, then the module is predicted

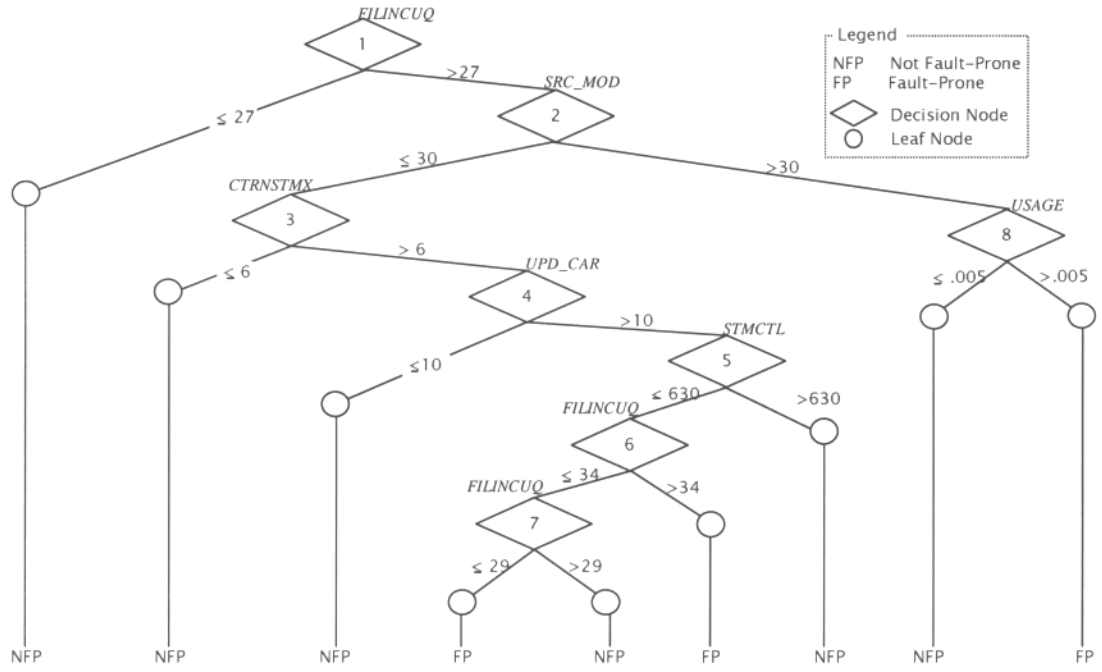


Figure 2: Classification tree case study

to be fault-prone. This models a non-linear non-monotonic relationship between file-includes and faults.

Table 10 lists the accuracy of the model in terms of Type I and Type II misclassification error rates. A Type I misclassification is when a module is actually not fault-prone, but the model predicts that the module is fault-prone. Conversely, a Type II misclassification is when a module is actually fault-prone, but the model predicts that the module is not fault-prone. The table shows results of three methods for estimating accuracy [11]: resubstitution, crossvalidation, and data-splitting. We believe that data-splitting gives the least biased estimate.

Other case studies by our research group have investigated the long-term usefulness of fault models, and found that a model trained with one release of a telecommunications

Table 10: Classification-tree accuracy

| | | Misclassification rates | |
|---|---|---|---|
| | $\zeta$ | Type I | Type II |
| Product & Process Metrics | 0.95 | | |
| Resubstitution (*fit*) | | 25.8% | 15.6% |
| Cross-validation (*fit*) | | 26.4% | 25.9% |
| Data Splitting (*test*) | | 26.2% | 28.9% |

system still had useful accuracy for three subsequent releases [16, 17].

## 5.2 Curvilinear Regression Case Study

**System description.** We performed a case study of data communications software using curvilinear regression to predict the number of faults [21]. The study examined previously published data [22] on a set of 226 data communications programs. The dependent variable was the number of errors in a program. The predictor was the number of noncomment lines of code, *LOC*.

**Curvilinear modeling.** The form of the model was the following.

$$y = b_0 + b_1 \, LOC^{b_2} + e \tag{1}$$

Gaffney argued that $b_2 = 4/3$ [5]. This study estimated $b_2 = 1.279$ which was significantly different from a linear model, i.e., $b_2 = 1$. Table 11 lists the estimated parameters for the two curvilinear models. See Appendix F for details on nonlinear regression, including curvilinear regression.

The analysis of variance in Table 12 shows that both models were significant at $\alpha < 0.05$. The analysis allocates total variation among the regression, and errors. The

Table 11: Curvilinear models

| Model | Parameter | Estimate | Std Dev |
|---|---|---|---|
| $LOC^{b_2}$ | | | |
| | $b_0$ | 0.81 | 0.14 |
| | $b_1$ | 0.002 | 0.00 |
| | $b_2$ | 1.279 | |
| $LOC^{4/3}$ | | | |
| | $b_0$ | 0.73 | 0.13 |
| | $b_1$ | 0.002 | 0.00 |

Table 12: Analysis of variance (ANOVA)

| Source | df | SS | MS | $F$ |
|---|---|---|---|---|
| $LOC^{b_2}$ | | | | |
| Reg | 1 | 232.37 | 232.37 | 93.65 |
| Error | 220 | 545.87 | 2.48 | |
| Total | 221 | 778.23 | | |
| $LOC^{4/3}$ | | | | |
| Reg | 1 | 442.22 | 442.22 | 171.98 |
| Error | 221 | 568.29 | 2.57 | |
| Total | 222 | 1010.51 | | |

table lists degrees of freedom (df), sum of squares (SS), mean of squares (MS), and the $F$ statistic.

Table 13 gives the accuracy of each model. A smaller average PRESS value means better accuracy. Thus, the model with the estimated exponent yielded slightly better quality of fit in this study.

Because the parameters of a nonlinear model are usually estimated to fit the data, the importance of nonlinearity is an empirical issue. Linear relationships may be adequate for some data sets. However, this case study illustrates that a nonlinear model is often preferred.

Table 13: Curvilinear-model accuracy

| Model | Avg PRESS | Adj $R^2$ |
|---|---|---|
| $LOC^{b_2}$ | 2.51 | 0.30 |
| $LOC^{4/3}$ | 2.66 | 0.44 |

## 5.3 Modeling

We advocate using a modeling method that is achieves the goals of the study and is appropriate for the characteristics of the data. Classification techniques yield predictions limited to fault-prone, or not. Regression techniques predict a quantity, such as the number of faults. In general, for the same data, classification is more powerful than regression because the result is less specific. However, management goals will determine which is required.

Table 14 lists selected modeling techniques for classification and for regression. Relevant appendices are noted (e.g., [B] refers to Appendix B). They are grouped into statistical techniques, tree-based techniques, and machine-learning techniques. Other supporting techniques are also listed. Note that curvilinear regression is a type of nonlinear regression. The Empirical Software Engineering Laboratory team at Florida Atlantic University has published case studies with each of the techniques listed. Table 15 lists general modeling tools that are useful for the modeling techniques listed in Table 14. One should note that these tools are readily available, supporting a wide variety of modeling techniques.

Table 14: Modeling Methods

| Classification | Regression |
|---|---|
| **Statistical** | |
| Logistic regression [B] | Multiple linear regression [E] |
| Discriminant analysis [C] | Curvilinear regression [F] |
| | Poisson regression [G] |
| **Tree-based** | |
| CART classification [D] | CART regression |
| S-Plus reg with class rule | S-Plus regression |
| TREEDISC (CHAID) | |
| **Machine learning** | |
| CBR classification | CBR interpolation |
| Neural net classification | Neural net function |
| Genetic programming | Genetic programming |
| **Supporting techniques** | |
| Principal components [A] | Principal components [A] |
| | Module-order modeling |

[Relevant appendix is noted]

Table 15: Example Modeling Tools

| | |
|---|---|
| SAS | |
| | Classical statistics |
| | TREEDISC |
| S-Plus | |
| | Classical statistics |
| | Tree-based models |
| CART | module in Systat |
| | Tree-based models |
| MATLAB | |
| | Neural networks |
| | Genetic programming |
| | Fuzzy sets |
| SMART | developed by FAU |
| | Case-Based Reasoning |
| | Module-order modeling |

# 6 Empirical validation must be realistic

Due to the many human factors that influence software reliability, controlled experiments to evaluate the usefulness of empirical models are not practical. Therefore, we take the case study approach to demonstrate their usefulness in a real-world setting. To be credible, the software engineering community demands that the subject of an empirical study be a system with the following characteristics [33]. The subject of a validation case study must be developed

- By a group, not by an individual
- By professionals, not by students
- In industry/government, not in laboratory
- As large as industry projects, not toy problem

Our case studies fulfill all of these criteria through collaborative arrangements with development organizations.

# 7 Conclusions

Development teams apply various techniques to improve software reliability. Due to resource and time constraints, one must often target reliability enhancement activities to high-risk modules. *Software quality models* predict which modules should be targeted.

Cost-effective software quality models must be developed through a disciplined methodology that balances accuracy with practical data collection. Drawing on experience with data from a variety of software development organizations, this report

presents the principles that are the foundation of our methodology, and reviews relevant case studies by the Empirical Software Engineering Laboratory at Florida Atlantic University. Collaboration between researchers and developers is the key to successful case studies.

# References

[1] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees.* Chapman & Hall, London, 1984.

[2] B. Efron. Estimating the error rate of a prediction rule: Improvement on cross-validation. *Journal of the American Statistical Association*, 78(382):316–331, June 1983.

[3] U. M. Fayyad. Data mining and knowledge discovery: Making sense out of data. *IEEE Expert*, 11(4):20–25, Oct. 1996.

[4] U. M. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. The KDD process for extracting useful knowledge from volumes of data. *Communications of the ACM*, 39(11):27–34, Nov. 1996.

[5] J. E. Gaffney, Jr. Estimating the number of faults in code. *IEEE Transactions on Software Engineering*, SE–10(4):459–464, July 1984.

[6] S. S. Gokhale and M. R. Lyu. Regression tree modeling for the prediction of software quality. In H. Pham, editor, *Proceedings of the Third ISSAT International Conference on Reliability and Quality in Design*, pages 31–36, Anaheim, CA, Mar. 1997. International Society of Science and Applied Technologies.

[7] D. J. Hand. Data mining: Statistics and more? *The American Statistician*, 52(2):112–118, May 1998.

[8] D. W. Hosmer, Jr. and S. Lemeshow. *Applied Logistic Regression.* John Wiley & Sons, New York, 1989.

[9] SAS. Institute. SAS/STAT software: The GENMOD procedure. Technical Report P-243, SAS Institute, Inc., Cary, NC, 1994.

[10] T. M. Khoshgoftaar and E. B. Allen. Multivariate assessment of complex software systems: A comparative study. In *Proceedings of the First International Conference on Engineering of Complex Computer Systems*, pages 389–396, Fort Lauderdale, Florida USA, Nov. 1995. IEEE Computer Society.

[11] T. M. Khoshgoftaar and E. B. Allen. Classification of fault-prone software modules: Prior probabilities, costs, and model evaluation. *Empirical Software Engineering: An International Journal*, 3(3):275–298, Sept. 1998.

[12] T. M. Khoshgoftaar, E. B. Allen, and J. Deng. Using regression trees to classify fault-prone software modules. Technical report, Florida Atlantic University, Boca Raton, Florida USA, Dec. 1999.

[13] T. M. Khoshgoftaar, E. B. Allen, R. Halstead, G. P. Trio, and R. Flass. Process measures for predicting software quality. *Computer*, 31(4):66–72, Apr. 1998.

[14] T. M. Khoshgoftaar, E. B. Allen, W. D. Jones, and J. P. Hudepohl. Return on investment of software quality models. In *Proceedings 1998 IEEE Workshop on Application-Specific Software Engineering and Technology*, pages 145–150, Richardson, TX USA, Mar. 1998. IEEE Computer Society.

[15] T. M. Khoshgoftaar, E. B. Allen, W. D. Jones, and J. P. Hudepohl. Data mining for predictiors of software quality. *International Journal of Software Engineering and Knowledge Engineering*, 9, 1999. Forthcoming.

[16] T. M. Khoshgoftaar, E. B. Allen, W. D. Jones, and J. P. Hudepohl. Accuracy of software quality models over multiple releases. *Annals of Software Engineering*, 6, 2000. Forthcoming.

[17] T. M. Khoshgoftaar, E. B. Allen, W. D. Jones, and J. P. Hudepohl. Classification tree models of software quality over multiple releases. *IEEE Transactions on Reliability*, 49(1), Mar. 2000. Forthcoming.

[18] T. M. Khoshgoftaar, E. B. Allen, K. S. Kalaichelvan, and N. Goel. Early quality prediction: A case study in telecommunications. *IEEE Software*, 13(1):65–71, Jan. 1996.

[19] T. M. Khoshgoftaar, E. B. Allen, X. Yuan, W. D. Jones, and J. P. Hudepohl. Assessing uncertain predictions of software quality. In *Proceedings of the Sixth International Software Metrics Symposium*, pages 159–168, Boca Raton, Florida USA, Nov. 1999. IEEE Computer Society.

[20] T. M. Khoshgoftaar, E. B. Allen, X. Yuan, W. D. Jones, and J. P. Hudepohl. Preparing measurements of legacy software for predicting operational faults. In *Proceedings: International Conference on Software Maintenance*, pages 359–368, Oxford, England, Aug. 1999. IEEE Computer Society.

[21] T. M. Khoshgoftaar and J. C. Munson. The lines of code metric as a predictor of program faults: A critical analysis. In *Proceedings: The Fourteenth Annual International Computer Software and Applications Conference*, pages 408–413, Chicago, Illinois USA, Oct. 1990. IEEE Computer Society.

[22] B. A. Kitchenham. An evaluation of software structure metrics. In *Proceedings of the International Computer Software and Applications Conference (COMPSAC)*, pages 369–376, Chicago, IL, 1988. IEEE Computer Society.

[23] P. A. Lachenbruch and M. R. Mickey. Estimation of error rates in discriminant analysis. *Technometrics*, 10(1):1–11, Feb. 1968.

[24] A. Mayer and A. M. Sykes. A probability model for analysing complexity metrics data. *Software Engineering Journal*, 4(5):253–258, Sept. 1989.

[25] J. A. Morgan and G. J. Knafl. Residual fault density prediction using regression methods. In *Proceedings of the Seventh International Symposium on Software Reliability Engineering*, pages 87–92, White Plains, NY, Oct. 1996. IEEE Computer Society.

[26] R. H. Myers. *Classical and Modern Regression with Applications.* Duxbury Series. PWS-KENT Publishing, Boston, 1990.

[27] S. L. Pfleeger, R. Jeffery, B. Curtis, and B. A. Kitchenham. Status report on software measurement. *IEEE Software*, 14(2):33–43, Mar. 1997.

[28] A. A. Porter and R. W. Selby. Empirically guided software development using metric-based classification trees. *IEEE Software*, 7(2):46–54, Mar. 1990.

[29] N. F. Schneidewind. Methodology for validating software metrics. *IEEE Transactions on Software Engineering*, 18(5):410–422, May 1992.

[30] G. A. F. Seber. *Multivariate Observations.* John Wiley and Sons, New York, 1984.

[31] D. Steinberg and P. Colla. *CART: A supplementary modules for SYSTAT.* Salford Systems, San Diego, CA, 1995.

[32] J. Troster and J. Tian. Measurement and defect modeling for a legacy software system. *Annals of Software Engineering*, 1:95–118, 1995.

[33] L. G. Votta and A. A. Porter. Experimental software engineering: A report on the state of the art. In *Proceedings of the Seventeenth International Conference on Software Engineering*, pages 277–279, Seattle, WA, Apr. 1995. IEEE Computer Society.

# A Principal Components Analysis

Software product metrics have a variety of units of measure, which are not readily combined in a multivariate model. We transform all product metric variables, so that each standardized variable has a mean of zero and a variance of one. Thus, the common unit of measure becomes one standard deviation.

Principal components analysis is a statistical technique for transforming multivariate data into orthogonal variables, and for reducing the number of variables without losing significant variation. Suppose we have $m$ measurements on $n$ modules. Let $\mathbf{Z}$ be the $n \times m$ matrix of standardized measurements where each row corresponds to a module and each column is a standardized variable. Our principal components are linear combinations of the $m$ standardized random variables, $Z_1, \ldots, Z_m$. The principal components represent the same data in a new coordinate system, where the variability is maximized in each direction and the principal components are uncorrelated [30]. If the covariance matrix of $\mathbf{Z}$ is a real symmetric matrix with distinct roots, then one can calculate its eigenvalues, $\lambda_j$, and its eigenvectors, $\mathbf{e}_j, j = 1, \ldots, m$. Since the eigenvalues form a nonincreasing series, $\lambda_1 \geq \ldots \geq \lambda_m$, one can reduce the dimensionality of the data without significant loss of explained variance by considering only the first $p$ components, $p \ll m$, according to some stopping rule, such as achieving a threshold of explained variance. For example, choose the minimum $p$ such that $\sum_{j=1}^{p} \lambda_j / m \geq 0.90$ to achieve at least 90% of explained variance.

Let $\mathbf{T}$ be the $m \times p$ standardized transformation matrix whose columns, $\mathbf{t}_j$, are defined as

$$\mathbf{t}_j = \frac{\mathbf{e}_j}{\sqrt{\lambda_j}} \text{ for } j = 1, \ldots, p \tag{2}$$

Let $D_j$ be a principal component random variable, and let $\mathbf{D}$ be an $n \times p$ matrix with $D_j$ values for each column, $j = 1, \ldots, p$.

$$D_j = \mathbf{Z}\mathbf{t}_j \tag{3}$$

$$\mathbf{D} = \mathbf{Z}\mathbf{T} \tag{4}$$

When the underlying data is software metric data, we call each $D_j$ a *domain metric.*

# B Logistic Regression

Logistic regression is a statistical modeling technique where the dependent variable, $Class_i$, has only two possible values [8]. Independent variables may be categorical, discrete, or continuous.

There are several possible strategies for encoding categorical independent variables. For binary categorical variables, we encode the categories as the values zero and one. Discrete and continuous variables may be used directly. Let $x_j$ be the $j^{th}$ independent variable, and let $\mathbf{x}_i$ be the vector of the $i^{th}$ module's independent variable values.

We designate a module being *fault-prone* as an "event". Let *fp* mean fault-prone and *nfp* mean not fault-prone. Let $\hat{p}(\mathbf{x}_i)$ be the estimated probability of an event, and thus, $\hat{p}(\mathbf{x}_i)/(1 - \hat{p}(\mathbf{x}_i))$ is the estimated odds of an event. The logistic regression model has the form

$$\log\left(\frac{\hat{p}(\mathbf{x}_i)}{1 - \hat{p}(\mathbf{x}_i)}\right) = b_0 + b_1 x_{i1} + \ldots + b_j x_{ij} + \ldots + b_m x_{im} \tag{5}$$

where log means natural logarithm and $m$ is the number of independent variables. Let $b_j$ be an estimated parameter. Logistic regression assumes that the probability of an event

is a monotonic function of each independent variable. The probability has the following form.

$$\hat{p}(\mathbf{x}_i) = \frac{\exp(b_0 + b_1 x_1 + \ldots + b_m x_m)}{1 + \exp(b_0 + b_1 x_1 + \ldots + b_m x_m)} \tag{6}$$

Given a list of candidate independent variables and a threshold significance level, $\alpha$, some of the estimated coefficients may not be significantly different from zero. Such variables should not be included in the final model. The process of choosing significant independent variables is called "model selection". *Stepwise logistic regression* is one method of model selection which uses the following procedure. Initially, estimate a model with only the intercept. Evaluate the significance of each variable not in the model. Add to the model the variable with the largest chi-squared $p$-value which is better than a given threshold significance level. Estimate parameters of the new model. Evaluate the significance of each variable in the model. Remove from the model the variable with the smallest chi-squared $p$-value whose significance is worse than a given threshold significance level. Repeat until no variables can be added or removed from the model. Tests for adding or removing a variable are based on an adjusted residual chi-squared statistic for each variable, comparing models with and without the variable of interest [8].

We calculate maximum likelihood estimates of the parameters of the model, $b_j$, using the iteratively reweighted least squares algorithm. Other algorithms are also available to calculate maximum likelihood estimates. This algorithm is used both in the stepwise procedure and for final estimates of model parameters. The estimated standard deviation of a parameter can be calculated, based on the log-likelihood function [26].

The odds ratio, $\psi_j$, is a statistic that indicates the relative effect on the odds of an

event by a one unit change in the $j^{th}$ independent variable [8]. For example, suppose $x_j$ is a binary variable with values zero or one. Let $p(1)$ be the probability of an event when $x_j = 1$, and let $p(0)$ be the probability of an event when $x_j = 0$, other things being equal.

$$\psi_j = \frac{p(1)/(1-p(1))}{p(0)/(1-p(0))} \tag{7}$$

Thus, the odds of an event for an observation with $x_j = 1$ is $\psi_j$ times the odds of an event for $x_j = 0$. The odds ratio is estimated by

$$\hat{\psi}_j = e^{b_j} \tag{8}$$

The odds ratio is a tool for interpretation of logistic regression models.

Given a logistic regression model, a module can be classified as *fault-prone* or not, by the following procedure: (1) Calculate $\hat{p}(\mathbf{x}_i)/(1-\hat{p}(\mathbf{x}_i))$ using Equation (5). (2) Assign the module by the following classification rule.

$$Class(\mathbf{x}_i) = \begin{cases} nfp & \text{if } \frac{1-\hat{p}(\mathbf{x}_i)}{\hat{p}(\mathbf{x}_i)} \geq \zeta \\ fp & \text{otherwise} \end{cases} \tag{9}$$

This rule enables a project to select its preferred balance between the misclassification rates by choosing a parameter $\zeta$. Given a candidate value of $\zeta$, we estimate misclassification rates, $\Pr(fp|nfp)$ and $\Pr(nfp|fp)$, by resubstitution of the *fit* data set into the model. If the balance is not satisfactory, we select another candidate value of $\zeta$ and estimate again, until we arrive at the preferred $\zeta$ for the project.

# C Nonparametric Discriminant Analysis

Nonparametric discriminant analysis is a statistical technique for predicting the class of an observation, such as the $i^{th}$ software module, represented by its vector of independent

variables, $\mathbf{x}_i$. Let *fp* mean the module is *fault-prone* and let *nfp* mean the modules is *not fault-prone.*

We use *stepwise* model selection at a significance level, $\alpha$, to choose the independent variables in the nonparametric discriminant model [30]. The candidate variables are entered into the model in an incremental manner, based on an $F$ test from analysis of variance which is recomputed for each change in the current model. Beginning with no variables in the model, the variable not already in the model with the best significance level is added to the model, as long as its significance is better than the threshold ($\alpha$). Then the variable already in the model with the worst significance level is removed from the model as long as its significance is worse than the threshold ($\alpha$). These steps are repeated until no variable can be added to the model.

We estimate a discriminant function based on the *fit* data set. Let $n_k$ be the number of observations in the class $k \in \{fp, nfp\}$. Let $\pi_k$ be the prior probability of membership in class $k$, which we usually choose to be the proportion of *fit* observations in each class. Let $f_k(\mathbf{x}_i)$ be the multivariate probability density giving the likelihood that a module represented by $\mathbf{x}_i$ is in class $k$. Since the density functions, $f_k$, are not likely to conform to the normal distribution, we use nonparametric density estimation. Let $\hat{f}_k(\mathbf{x}_i|\lambda)$ be an approximation of $f_k(\mathbf{x}_i)$, where $\lambda$ is a smoothing parameter in this context. Let $\mathbf{S}_k$ be the covariance matrix for all observations in class $k$, and $|\mathbf{S}_k|$ is its determinant. Let $K_k(\mathbf{u}|\mathbf{v}, \lambda)$ is a multivariate kernel function on vector $\mathbf{u}$ with modes at $\mathbf{v}$. We select the normal kernel.

$$K_k(\mathbf{u}|\mathbf{v}, \lambda) = (2\,\pi\lambda^2)^{-n_k/2}|\mathbf{S}_k|^{-1/2}\exp(\,(-1/2\lambda^2)(\mathbf{u}-\mathbf{v})'\mathbf{S}_k^{-1}(\mathbf{u}-\mathbf{v})\,) \tag{10}$$

Let $\mathbf{x}_{kl}, l = 1, \ldots, n_k$ represent a module in class $k$. The estimated density function is

given by the multivariate kernel density estimation technique [30].

$$\hat{f}_k(\mathbf{x}_i|\lambda) = \frac{1}{n_k} \sum_{l=1}^{n_k} K_k(\mathbf{x}_i|\mathbf{x}_{kl}, \lambda) \tag{11}$$

We empirically choose a preferred value for $\lambda$ based on results from cross-validation using the *fit* data set.

A classification rule that minimizes the expected number of misclassification is the following.

$$Class(\mathbf{x}_i) = \begin{cases} nfp & \text{if } \frac{\hat{f}_{nfp}(\mathbf{x}_i|\lambda)}{\hat{f}_{fp}(\mathbf{x}_i|\lambda)} \geq \frac{\pi_{fp}}{\pi_{nfp}} \\ fp & \text{otherwise} \end{cases} \tag{12}$$

A generalized classification rule is the following.

$$Class(\mathbf{x}_i) = \begin{cases} nfp & \text{if } \frac{\hat{f}_{nfp}(\mathbf{x}_i)}{\hat{f}_{fp}(\mathbf{x}_i)} \geq \zeta \\ fp & \text{otherwise} \end{cases} \tag{13}$$

where $\zeta$ is a parameter which we choose to achieve a preferred balance between misclassification rates.

# D Classification And Regression Trees (CART)

The Classification And Regression Trees (CART) algorithm [1] builds a classification tree. It is implemented as a supplementary module for the SYSTAT package [31]. We follow terminology in the classification tree statistics literature, calling independent variables "predictors". We model each module with a set of continuous ordinal-scaled predictors, such as, software product and process metrics, and a nominal-scaled dependent variable (a "response" variable) with two categories, *not fault-prone* (*nfp*) or *fault-prone* (*fp*).

Beginning with all modules in the root node, the algorithm recursively partitions ("splits") the set into two leaves until a stopping criterion applies to every leaf node.

A goodness-of-split criterion is used to minimize the heterogeneity ("node impurity") of each leaf at each stage of the algorithm. Further splitting is impossible if only one module is in a leaf, or if all modules have exactly the same measurements. CART also stops splitting if a node has too few modules (e.g., less than 10 modules). The result of this process is typically a large tree. Usually, such a maximal tree overfits the data set, and consequently, is not robust. CART then generates a series of trees by progressively pruning branches from the maximal tree. The accuracy of each size of tree in the series is estimated and the most accurate tree in the series is selected as the final classification tree.

Early work with classification trees in software engineering selected branches by a measure of homogeneity until a stopping rule was satisfied [28]. More recent work has used an algorithm based on deviance [12, 32], and an algorithm based on statistical significance [19, 20]. CART's default goodness-of-split criterion is the "Gini index of diversity" which is based on probabilities of class membership [1].

*Resubstitution* is a method for estimating model accuracy by using the model to classify the same modules that were the basis for building the model, and then calculating misclassification rates. The estimated accuracy can be overly optimistic.

*$\nu$-fold cross-validation* is an alternative method that also uses the same modules as were the basis for building the model, but the estimated accuracy is not biased [2, 6, 23]. The algorithm has these steps: Randomly divide the sample into $\nu$ approximately equal subsets (e.g. $\nu = 10$). Set aside one subset as a test sample, and build a tree with the remaining modules. Classify the modules in the test subset and note the accuracy of each prediction. Repeat this process, setting aside each subset in turn. Calculate the overall accuracy. This is an estimate of the accuracy of the tree built using all the modules.

The number of subsets, $\nu$, should not be small; Breiman, et al. found that ten or more worked well [1].

CART allows one to specify prior probabilities, and costs of misclassifications. These parameters are used to evaluate goodness-of-split of a node as a tree is recursively generated. Let $\pi_{fp}$ and $\pi_{nfp}$ be prior probabilities of membership in the *fault-prone* and *not fault-prone* classes, respectively, and let $C_I$ and $C_{II}$ are the costs of Type I and Type II misclassifications, respectively.

Due to different costs associated with each type of misclassification, we need a way to provide appropriate emphasis on Type I and Type II misclassification rates according to the needs of the project. We experimentally choose a parameter $\zeta$, which can be interpreted as a priors ratio times a cost ratio.

$$\zeta = \left(\frac{\pi_{fp}}{\pi_{nfp}}\right)\left(\frac{C_{II}}{C_I}\right) \tag{14}$$

We have observed a tradeoff between the Type I and the Type II misclassification rates, as functions of $\zeta$. Generally, as one goes down, the other goes up. We estimate these functions by repeated calculations with the *fit* data set. Given a candidate value of $\zeta$, we estimate prior probabilities $\pi_{fp}$ and $\pi_{nfp}$ by proportions in the fit data set; we set $C_I = 1$, and we choose $C_{II}$ to achieve the candidate value of $\zeta$. We build a tree and estimate the Type I and Type II rates using resubstitution and $\nu$-fold cross-validation. We repeat for various values of $\zeta$, until we arrive at the preferred $\zeta$ for the project.

# E Multiple Linear Regression

Even though we may have a long list of independent variables, it is possible that some do not significantly influence the dependent variable. If an insignificant variable is included in the model, it may add noise to the results and my cloud interpretation of the model. For example, if a coefficient, $b_j$ for the $j^{th}$ variable in a linear model is not significantly different from zero, then it is best to omit that term from the model. The process of determining which variables are significant is called *model selection.* Of several model selection techniques available for multiple linear regression, we use the *stepwise regression* method [26], and the *fit* data set. Stepwise model selection is an iterative procedure. All the candidate independent variables are specified. Significant variables are added and insignificant variables are removed from the model on each iteration, based on an $F$ test. The test is recomputed on each iteration, until no variable can be added to or removed from the model.

Many models have a general mathematical form with parameters that must be chosen so that the *fit* data set matches the model as closely as possible. This step consists of estimating the values of such parameters. Suppose there are $n$ observations in the *fit* data set, and the subscript $i$ indicates data for the $i^{th}$ observation. In general, a multivariate linear model has the following form.

$$\hat{y}_i = b_0 + b_1 x_{i1} + \ldots + b_m x_{im} \tag{15}$$

$$y_i = b_0 + b_1 x_{i1} + \ldots + b_m x_{im} + e_i \tag{16}$$

where $x_{i1}, \ldots, x_{im}$ are the independent variables' values, $b_0, \ldots, b_m$ are parameters to be estimated, $\hat{y}_i$ is the predicted value of the dependent variable, $y_i$ is the dependent variable's actual value, and $e_i = y_i - \hat{y}_i$ is the error for the $i^{th}$ observation. We estimate

the parameters, $b_0, \ldots, b_m$, using the *least squares* method. This method chooses a set of parameter values that minimizes $\sum_{i=1}^{n} e_i^2$ [26].

When the parameters have been estimated, and given each set of independent variable values, a model can calculate a value of the dependent variable. Since the independent variables are known earlier than the actual value of the dependent variable, the calculated value is a *prediction.*

# F Nonlinear Regression

Many models have a general mathematical form with parameters that must be chosen so that the *fit* data set matches the model as closely as possible. This step consists of estimating the values of such parameters. Suppose there are $n$ observations in the *fit* data set, and the subscript $i$ indicates data for the $i^{th}$ observation. In general, a nonlinear model has the following form.

$$\hat{y}_i = f(\mathbf{x}_i, \mathbf{b}) \tag{17}$$

$$y_i = f(\mathbf{x}_i, \mathbf{b}) + e_i \tag{18}$$

where $\mathbf{x}_i$ is a vector of the independent variables' values, $\mathbf{b}$ is a vector of parameters to be estimated, $\hat{y}_i$ is the predicted value of the dependent variable, $y_i$ is the dependent variable's actual value, and $e_i = y_i - \hat{y}_i$ is the error for the $i^{th}$ observation. We estimate the parameters, $\mathbf{b}$, using the *least squares* method. This method chooses a set of parameter values that minimizes $\sum_{i=1}^{n} e_i^2$ [26].

In the nonlinear situation, a closed-form solution for $\mathbf{b}$ generally does not exist. Thus, we use an iterative process to find a satisfactory set of parameters. The iterative

process starts at some candidate set of parameter values, $\mathbf{b}_0$, and then use the data to compute a new set of candidate parameters that improves the squared error. There are several iterative methods available, including steepest descent which uses the gradient, Gauss-Newton which uses a Taylor series, Newton which uses the second derivative, and Marquardt which combines the Gauss-Newton and steepest descent methods. Ordinarily, these methods will converge on the same parameters.

# G Poisson Regression

Multiple linear regression makes certain assumptions about the data, such as the dependent variable, $y$, is continuous, or the variance of the residual error is *homogeneous*, i.e., independent of $y$. These assumptions are often violated by software engineering data. For example, the number of faults is a commonly studied software quality metric; its values are integers. Violating assumptions may result in a multiple linear regression model with limited predictive accuracy or unwarranted conclusions. Poisson regression is an alternative technique that is sometimes appropriate when multiple linear regression is not. See Myers [26] and SAS documentation [9] for details.

Poisson regression is applicable when the dependent variable, $y$, takes only integer values, and the variance of the residual error may depend on $y$. It is based on the Poisson probability distribution.

$$\Pr(y) = \frac{(\lambda t)^y}{y!} e^{-\lambda t} \tag{19}$$

where $\lambda$ is a parameter, $t$ is "time", and $\lambda t = \mathrm{E}(y) = \mathrm{Var}(y)$, the expectation and the variance of $y$.

Poisson models are often used in reliability studies where $y(t)$ is the number of

failures experienced up to time $t$. In this context, $\lambda$ is interpreted as the average failure rate. In a homogeneous Poisson model, $\lambda$ is a constant. In a nonhomogeneous model, it is a function of time.

A Poisson regression models a multiplicative relationship between $\lambda$ and independent variables, $x_j, j = 1, \ldots, m$.

$$\lambda = e^{\beta_0} e^{\beta_1 x_1} \ldots e^{\beta_m x_m} \tag{20}$$

$$\ln \lambda = \beta_0 + \beta_1 x_1 + \ldots + \beta_m x_m \tag{21}$$

Let $\boldsymbol{\beta}$ be the vector of parameters, and let $\mathbf{b}$ be the vector of their estimated values. Let $\mathbf{x}$ be the vector of independent variables for an observation, and let $\mathbf{x}'$ be its transpose. Then the matrix notation for Equation (21) is the following.

$$\ln \lambda = \mathbf{x}'\boldsymbol{\beta} \tag{22}$$

Given a set of $n$ observations, the parameters, $\boldsymbol{\beta}$, can be estimated from Equation (19) and (22) using maximum likelihood techniques.

$$\Pr(y_i|\boldsymbol{\beta}) = \frac{(t_i \exp(\mathbf{x}_i'\boldsymbol{\beta}))^{y_i}}{y_i!} \exp(-t_i \exp(\mathbf{x}_i'\boldsymbol{\beta})) \tag{23}$$

where $i = 1, \ldots, n$. Given the estimated parameters, $\mathbf{b}$, and a data point, $(t_i, \mathbf{x}_i)$, The predicted value of $y_i$ is the following.

$$\hat{y}_i = t_i \exp(\mathbf{x}_i'\mathbf{b}) \tag{24}$$

Let $\mathbf{y}$ be the vector of observed $y_i$ and $\hat{\mathbf{y}}$ be the corresponding vector of predicted $\hat{y}_i$ for the *Fit* data set. Let $\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})$ be their log likelihood function. $\mathcal{L}(\mathbf{y}, \mathbf{y})$ represents a perfect fit. Quality of fit is measured by deviance, defined as

$$D(\mathbf{y}, \hat{\mathbf{y}}) \equiv 2(\mathcal{L}(\mathbf{y}, \mathbf{y}) - \mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})) \tag{25}$$

$D$ has a $\chi^2$ distribution with $(n-(m+1))$ degrees of freedom.

A perfect Poisson model has mean equal to its variance. Unfortunately, the estimated variance of a Poisson regression is often larger than the mean. This is modeled by a constant *dispersion* factor, $\phi$, where $\mathrm{Var}(y) = \phi \mathrm{E}(y)$. The following is an estimate for $\phi$.

$$\hat{\phi} = \frac{D}{n-(m+1)} \tag{26}$$

**Model Selection.** Suppose we are considering $p$ independent variables. *Model selection* is the process of identifying which of these are significantly related to the dependent variable. In other words, we want to identify those variables where we are confident that their coefficients are nonzero.

Let $D_0$ be the deviance of a model with $m$ variables, and let $\phi$ be its dispersion factor. Let $D_1$ be the deviance of a submodel that has one less variable. The asymptotic distribution of $F = (D_1 - D_0)/\hat{\phi}$ is the $F$ distribution with 1 and $(n-m)$ degrees of freedom, assuming that $\hat{\phi}$ is well behaved. *Type 1* analysis uses $F$ to evaluate the significance of the variables [9].

Type 1 analysis fits a sequence of models, beginning with one that only has an intercept term. Each model in the sequence has one additional variable, until all candidate variables have been considered. When defining $F$, Model 0 is the current model and Model 1 is the prior model. At each step, we test the null hypothesis that the coefficient of the new variable is zero using the $F$ statistic.

Since this process depends on the order that the variables are considered, we want to arrange the variables so that those with a large $F$ statistic are early in the list and smaller ones are at the end. Therefore, we experiment with the order of the variables, and make a Type 1 analysis for each order. When we find an order where all of the

insignificant variables are grouped at the end, we make our model selection, choosing the significant variables at the beginning of the list.

**Identifying Outliers.** Let $l(y_i, \hat{y}_i)$ be the individual contribution of the $i^{th}$ observation to the log likelihood function $\mathcal{L}(\mathbf{y}, \hat{\mathbf{y}})$. A deviance residual for the $i^{th}$ observation, $r_{Di}$, is defined by the following.

$$d_i = 2(l(y_i, y_i) - l(y_i, \hat{y}_i)) \quad (27)$$

$$D = \sum_{i=1}^{n} d_i \quad (28)$$

$$r_{Di} = \text{sign}(y_i - \hat{y}_i)\frac{\sqrt{d_i}}{\sqrt{\phi}} \quad (29)$$

The $\sqrt{\phi}$ in the denominator compensates for over dispersion. $r_{Di}$ is approximately normally distributed, and thus, a double tailed hypothesis test can identify outliers [24].

**Removing Insignificant Variables.** After removing outliers, it is possible that one or more of the variables that were significant during model selection now have coefficients that are approximately zero, i.e., are now insignificant. *Type 3* analysis is useful to identify such variables [9].

Type 3 analysis uses the same $F$ statistic as Type 1 analysis. Model 0 is the full model and Model 1 is similar with one less variable. The $F$ statistic can then be used to evaluate whether the variable is still significant. Each variable is considered independently, so that Model 0 is always the same full model.

We calculate a Type 3 analysis for each variable in the model after outliers have been removed. If a variable is now insignificant, we remove it from the model. In other words, we consider its coefficient to be zero.